Programming Project #10

Assignment Overview
In this assignment you will continue your practice creating a combination data structure, `MapSet`, which combines a map and a set. We change the specifications so that you cannot use a STL vector nor pair (as well as not use map or set or the sort algorithm). You will update your class to use **dynamic arrays**. It is due 04/16, Monday, before midnight on Mimir. Project is worth 60 points (6% of your overall grade).

Background
We are going to stay with the `MapSet` problem but update its implementation. We are going to stop using vectors and pairs and do the work ourselves with an underlying dynamic array. We will also template to the class to work with different types.

We do this as a better way to get familiar with the underlying problem of templated classes and dynamic arrays. The specs of the `MapSet` should be familiar to you now, so it is a matter of changing the implementation. The focus will be on this implementation.

Details
As this is a templated class you can ***only*** write a header file to solve the problem. Thus, we will provide a proto-header file, `proj10_mapset.h`, which you will fill in. In the specifications below, we will focus on what is different first for this vs. the previous `MapSet`, then update the rest of the already familiar set of methods.

Class Node
Since we cannot use the STL pair, we are going to have to write our own `struct` called `Node`. This will take the place of the `pair` from the STL. Here is the header part of Node:

```
template<typename K, typename V>
   struct Node {
   K first;
   V second;
   Node() = default;
   Node(K,V);
   bool operator<(const Node&) const;
   bool operator==(const Node&) const;
   friend ostream& operator<<(ostream &out, const Node &n){
      // YOUR CODE HERE
   }
};
```

First note it is doubly templated on the Key (K) and the Value (V). Conveniently we name the data members `first` and `second` to be compatible with `std::pair`. You are required to write in this header file:

- 2 arg constructor
- `operator<` Compares two `Node` instances based on the first values (so that, if we need to compare using something like `lower_`bound, they would be ordered by key, which is what we want).
- `operator==` Similarly, two `Node` instances are equal if their two `first` values are equal (can't have duplicate keys).
- `operator<<` As discussed in the videos, and unlike the other definitions (which should appear below the `struct`), the easiest way to do `operator<<` for a templated class is to place the definition ***right here*** in the class section (not below where the other methods would go). It should print "`first:second`" to the `ostream`, whatever the values of `first` and `second` are.

Class MapSet
The underlying data member for the storage of `Node` is an array (not a vector, a basic array) of type `Node`. Since this is not a vector we have to grow the vector when we add more `Node` elements than the array can presently store. The array has to grow dynamically during the course of the run. As before, the elements of the array must be in sorted order of the Nodes, using the key/first as the element to sort by. Here's the header part:

```
template<typename K, typename V>
class MapSet{
 private:
    Node<K,V>* ary_;
    size_t last_;
    size_t capacity_;
    Node<K,V>* find_key(K);
    void grow ();
 public:
     MapSet(int sz = 2);
     MapSet(initializer_list< Node<K,V> >);
     MapSet (const MapSet&);
     MapSet operator=(MapSet);
     ~MapSet();
     size_t size();
     bool remove (K);
     bool add(Node<K,V>);
     Node<K,V> get(K);
     bool update(K,V);
     int compare(MapSet&);
     MapSet mapset_union (MapSet&);
     MapSet mapset_intersection(MapSet&);

     friend ostream& operator<<(ostream &out, const MapSet &ms){
        // YOUR CODE HERE
     }
 };
```

Data Members

- `capacity_` the actual size of the underlying array. It is the number of the `Node` elements the array **can hold** before it must grow.
- `last_` the index of the first open (unsued) location in the array. When `last_ == capacity_` then the array must grow.

Methods new to Project 10

- `1 arg constructor` (default value for size). Creates an array of size `sz` (the parameter) of type `Node` using the `new` operator. The `sz` parameter also is the is the `capacity_` of the `MapSet`. `last_` is set to 0 (the first open index).
- `grow` . This method doubles the capacity of the underlying array. It:
    - create a new array of twice the capacity
    - copies the content of the old array into the new
    - update `capacity_` and `ary_`, delete the old array
- `copy constructor`
- `operator=`
- `destructor`
  These are all methods that we have ignored previously because we took the defaults for those methods. That worked because their operations were taken care of by the underlying STL elements. Now you must provide them yourself. You have to **<u>look at the example code</u>**, videos and notes to create these. In fact, **copy the example code an modify to suit your needs here** (you won't get accused of cheating for this, I promise).

The destructor is tested in that memory leaks are tested (Test 35). Other than that these are tested implicity in the code but YOU should test them yourself!

Methods modified for Project 10

- `initializer_list constructor` Create an array of size `initializer_list.size().` Copy each `Node` from the list and place in the array. The `initializer_list` does not have to be in sorted order but the array should be sorted after you add all the elements. (Hint: write `add` first and use it here)
- `Node<K,V>* find_key(K key)` As before this is a private method only usable by other `MapSet` methods (not from a main program), but it now returns a pointer to a `Node<K,V>` (not an iterator). You can still use `lower_bound` if you do pointer arithmetic, though the `begin()` and `end()` functions won't work as the array size is not fixed at compile time. The pointer is either the first `Node` in the array that is equal to (by key) or greater than the key, or `nullptr` (the last two meaning that the key isn't in `ary_`). It must be private because `ary_` is private and we cannot return a pointer to private data.

  To make `lower_bound` work, you must have an `operator<` for `Node`. But you have to write that so it works nicely

  This function is ***<u>not tested</u>*** in the Mimir test set but necessary everywhere. However, it is essentially the use of `lower_bound`. It is a good to isolate it however for future projects.

- `size()` : size of the `MapSet` (number of `Nodes`)
- `get(K)` : returns a `Node<K,V>` that is either a copy of the `Node` that has the string as a key or a pair with default values.
- `update(K, V)` : if the key is in the `MapSet`, update the key-value pair to the value. Return true. If the key is not in `MapSet`, do nothing and return false.
- `remove(string)` : if the key is in the `MapSet`, remove the associated `Node` and return true. If the key is not in the `MapSet` do nothing and return false.
- `add(string,long)` : if the is in the `MapSet`, do nothing and return false. Otherwise create a Node with the argument values and insert the new pair into the array, ***<u>in sorted order</u>***, and return true.
- `compare(MapSet&)` : compare the two `MapSets` lexicographically, that is element by element using the key of the Nodes as comparison values. If you compare two `Node`, then the comparison is based on the `.first` of each pair (that is, the string-key of each pair, which is what `operator==` of Node does)If the argument `MapSet` is greater, return -1. If all of the comparable pairs are equal but one `MapSet` is larger (has more Nodes), then the longer determines the return value (1 if the first is longer, -1 if the second).
- `mapset_union(MapSet&)`. Return a new `MapSet` that is a union of the two `MapSets` being called. Again, comparison on whether an element is in the `MapSet` is based on the key. There is an order here. If the two `MapSets` have the same key but different values, then the key-value of the calling `MapSet` is the one that is used.
- `mapset_intersection(MapSet&)`. Return a new `MapSet` that is the intersection of the two `MapSets` being called. Again, comparison on whether an element is in the `MapSet` is based on the key. There is an order here. If the two `MapSets` have the same key but different values, then the key-value of the calling `MapSet` is the one that is used.
- `friend ostream& operator<<(ostream&, MapSet&)`. Returns the ostream after writing the `MapSet` to the `ostream`. The formatting should have each pair colon (':') separated, and each pair comma + space separated (', '). E.g., `Ann:1234, Bob:3456, Charlie:5678` for `Node<string,long>`

Requirements

We provide `proj10_mapset.h` as a skeleton that you must fill in. You submit to Mimir `proj10_mapset.h`

We will test your files using Mimir, as always.

Deliverables
`proj10/proj10_mapset.h`

1. Remember to include your section, the date, project number and comments.
2. Please be sure to use the specified directory and file name.

Assignment Notes

lower_bound
You should get how `lower_bound` works now. The difference here is now you have no iterators to work with. You must instead use pointer arithmetic.

```
lower_bound(ary_,  ary_+last_, value_to_search_for)
```

Conveniently, there is an `operator<` for `Nodes` so no binary predicate is required.

The return value is a pointer (not an iterator) to the either the element in the container that meets the criteria, or the value of the last element in the range searched (in this case, `nullptr`)

That means that either:

- the `value_to_search_for` is already in the container and the iterator points to it.
- `value_to_search_for` is not in the container. Not in the container means:
    - the iterator points to a value "just greater" than the `value_to_search_for`
    - the iterator points to `ayr_+last_`

array insert or erase
Bad news, there is no insert nor erase. You can write these separately or put them in add and remove (respectively). To make these work, you are going to have to remake the arrays.

- for insert: make a new array that has: copy of the old array up to to insert point + new element + the old array after the insert point.
    - if you made an array with new, better delete it or you will leak!
- for erase: make a new array that has: copy of old array up to the remove point, then copy after the old array to the end. In other words, skip the element being erased in the copy.

add
The critical method is `add`. Get that right first and them much of the rest is easy. For example, the initializer list constructor can then use `add` to put elements into the vector at the correct location (in sorted order).

set_union and **set_intersection**
Do it like you did last time, but use pointers instead of iterators.

sort
As before, no use of sort allowed. If you use sort in a test case you will get 0 for that test case. Do a combination of lower_bound and vector insert to get an element where it needs to be in a vector.